Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 65 (2007) 222–248
Science of Computer Programming
www.elsevier.com/locate/scico

Visual assessment of software evolution

Lucian Voinea*, Johan Lukkien, Alexandru Telea

Department of Computer Science, Technische Universiteit Eindhoven, The Netherlands

Received 1 December 2005; received in revised form 30 March 2006; accepted 15 May 2006
Available online 20 February 2007

Abstract

Configuration management tools have become well and widely accepted by the software industry. Software Configuration Management (SCM) systems hold minute information about the entire evolution of complex software systems and thus represent a good source for process accounting and auditing. However, it is still difficult to use the entire spectrum of information such tools maintain. Currently, significant effort is being done in the direction of mining this kind of software repositories for extracting data to support relevant assessments. In this article we propose a concerted set of visualization tools and techniques for the assessment of software evolution based on the information stored into SCM systems. Firstly, we introduce a generic way to obtain models of source code at different levels of detail and from different perspectives. Secondly, we propose a set of visual representations and techniques to efficiently and effectively depict the evolution of these code models. These techniques target specific questions and assessments, from the detailed code developer perspective to the overview required by system architects and project managers. We detail the concrete implementation of two such code models and corresponding visual representations. The file view describes code change at line level across multiple versions of a single file, or small number of files. The project view shows changes at file level across complete software projects. All our views share the same visual and interactive techniques, enabling users to easily switch among and correlate between them. We implement our visual techniques to quickly and compactly display and navigate the evolution of tens of thousands of artifacts on a single screen. We demonstrate our techniques with several use cases performed on real world, industry-size code bases and outline the concrete findings and ways our visualizations helped in understanding various types of code changes.

Keywords: SCM data mining; Software evolution; Software visualization

1. Introduction

Software Configuration Management (SCM) systems are an essential ingredient of effectively managing large-scale software development projects. Due to the growing complexity and size of industry projects, management systems that automate, help and/or enforce a specific development, testing and deployment process, have become a "must have" [6].

One of the main characteristics of a SCM system is that it maintains a history of changes done in the structure and contents of the managed project. This serves primarily the very precise goal of navigating to and retrieving a specific version in the evolution of the project. However, SCM systems and the information they maintain enable

* Corresponding author.
E-mail addresses: l.voinea@tue.nl (L. Voinea), j.j.lukkien.@tue.nl (J. Lukkien), alext@win.tue.nl (A. Telea).

0167-6423/$ - see front matter © 2007 Published by Elsevier B.V.
doi:10.1016/j.scico.2006.05.012

also a large variety of possibilities that fall outside the above very precise goal. The intrinsically maintained system evolution information is a very convenient starting point for empirically understanding the software development process and structure. One of the main areas that can benefit from this information is the software maintenance of large projects.

Industry surveys show that, in the last decade, maintenance and evolution exceeded 90% of the total software development costs [12], a problem referred to as the *legacy crisis* [33]. It is, therefore, of paramount importance to bring these costs down. This challenge is addressed on two fronts, as follows. The *preventive* approach tries to improve the overall quality of a system upfront, at design time. Many tools and techniques exist to assess and improve the design-time quality attributes [11,21]. However, the sheer dynamics of the software construction process, its high variability, and the quick change of requirements and specifications make such an approach either cost-ineffective or even inapplicable in many cases. Increasingly popular software development methodologies, such as extreme programming [4], explicitly acknowledge the high dynamics of software and thus fit the preventive approach to a very limited extent only. The *corrective* approach aims to facilitate the maintenance phase itself, and is supported by program and process understanding and fault localization tools (e.g. [10,23,38]). In most projects, however, it is often the case that appropriate documentation does not exist, or it is 'out of sync' with the implementation. In such cases, the code evolution information maintained on a SCM system (assuming such a system is used) is the one and only up-to-date, definitive reference material available. Efficiently exploiting this information can greatly help the maintainers to understand and manage the evolving project.

Many tools have been designed to help in revealing the structure of software systems starting from source code (e.g. [10,36,38,39]). Most such tools focus on visualizing high-level software abstraction, such as classes, modules, and packages, extracted from source code in a reverse engineering process. However, such tools do not show lower-level software changes, such as the many, minute source code edits done during debugging. Moreover, such tools often focus on a fixed system structural view that does not show all changes the code has undergone with time. Various graph-drawing techniques, such as the one proposed by Collberg et al. [7], have tried to overcome this limitation by showing also the temporal dimension of software structure and mechanism evolution. However, this still-to-be-validated approach does not seem to scale well on real-life code bases. Moreover, the visual graph representations used are often perceived as too abstract by the actual developers who think in terms of the source code itself. At the other end of the granularity spectrum, the SeeSoft tool [10] uses a line-based visual approach (see also [3]). Source files are seen as a set of code lines, each of which is drawn as a pixel line. This allows visualizing many thousands of lines on a single screen. Several such techniques and tools have been proposed, such as Aspect Browser [20], Bee/Hive [32], sv3D [27] and Augur [15]. While these approaches succeed in revealing structure and change dependencies between code fragments, they only offer snapshots in time, and do not reveal changes in the global context of an entire project life span. A global overview would allow the discovery of problems in a specific part of the code that appear after another part was changed. This kind of insight is easier to get when visualizing the context of an entire project evolution. In contrast, intensive debugging and runtime analysis is needed to get it from a single code snapshot. Another, useful case of global investigations is to identify the files that contain tightly coupled implementations. Such files can be easily identified in a global context as they most likely have a similar evolution. In contrast, detailed manual cross-file analysis has to be performed in order to achieve the same result using a static system visualization.

In this paper, we propose and discuss a set of new techniques for visually assessing the entire evolution of software projects using the evolution information contained in SCM systems. Typical questions we try to answer with our techniques are:

- What code was added, removed, or altered and when?
- Where did the changes take place?
- Who performed these modifications of the code?
- Which parts of the code are unstable?
- How are source code changes correlated?
- How are the development tasks distributed among the programmers?
- What is the context in which a piece of code appeared?
- What are the project files that belong and/or are modified together?

We validated our techniques by implementing them in a toolset that seamlessly combines SCM data extraction with analysis and visualization. We present in this paper the results of using our toolset for the assessment of several

industry-size software projects: the FreeBSD Linux distribution [14], the ROBOCOP component framework [22], and the VTK library [44].

In Section 2 we overview existing efforts in analyzing the evolution information encapsulated in SCM systems. Section 3 gives a formal description of the software evolution data that we explore using visual means. Section 4 presents the visual techniques and methods we propose for the assessment of evolution. In Sections 5–7 we present three user studies we performed to validate the proposed techniques, and detail the types of questions our toolset was able to answer. Section 8 reflects on the open issues and possible ways to address them.

2. Background

The huge potential of the information stored on SCMs, as a starting point for empirical studies on software development, has been only recently acknowledged by the research community. The massive growth in popularity and spread of SCM systems, greatly influenced by open source projects like CVS [9] and Subversion [37], opened new possibilities in the areas of project accounting, auditing and understanding. The efforts have been concentrated so far in two directions of research: data mining and data visualization.

Data mining research focuses on processing and extracting relevant information from the evolution data stored into SCM systems. However, these systems have not been designed with support for empirical studies in mind, so they often lack direct access to important evolution information. Most of this kind of information is usually inferred from the "raw" information that the SCMs contain by the data mining tools themselves. Many methods have been proposed to offer access to higher level, aggregated information about the project evolution. Fischer et al. propose in [13] a novel method to extend the evolution data contained in the SCMs with information about file merge points. Additionally, they present the benefits of integrating SCM evolution data with specific information about bug tracking. Sliwerski et al. [35] propose a similar integration to predict the introduction of defects in the code. Recovering of SCM transactions has been more extensively addressed. Gall [16], German [18] and Mockus [28] propose transaction recovery methods based on fixed time windows. Zimmermann and Weigerber built on this work, and propose in [50] better mechanisms that involve sliding windows and information acquired from the commit mails. Ball proposes in [2] a new metric for class cohesion based on the SCM extracted probability of classes being modified together. Relations between classes based on change similarities have been proposed also by Bieman *et al.* [5] and Gall et al. [16]. Relations between finer grained building blocks, like functions, are also addressed by Zimmermann et al. in [48, 49] and by Ying et al. in [45]. Lopez-Fernandez *et al.* [26] apply general social network analysis methods on the information stored in SCMs to characterize the development process of industry size projects and find similarities between them. Ohira et al. [30] exploit the user information stored in SCMs to build cross process social networks for easy sharing of knowledge.

Data visualization, the second research direction, takes a different path. Here, the focus is on how to make the large amount of evolution information effectively available to the user. Data visualization techniques make few or no assumptions about the data — the goal is to let the user discover patterns and trends by himself rather than coding pattern models to be searched for in the mining process. Consequently, visualization tools strive to present the data in a form that is as intuitive and familiar as possible to users. SeeSoft [10] is the first tool that uses a direct 'code line to pixel line' visual mapping. Color is used to show code fragments that correspond to a given modification request. The Aspect Browser [20] uses regular expressions to locate specific artifacts (e.g. key words) and then it visualizes their distribution. Augur [15], combines in a single visual frame information about both artifacts and activities of a software project at a specific moment. The above tools are successful in revealing the line-based structure of software systems, and uncover code line-level change dependencies at given moments in time. However, they do not provide insight into code attributes and structural changes made throughout an entire project duration.

As a first step in this direction, UNIX's `gdiff` and its Windows version `WinDiff` visualize code differences between only two versions of a given file by depicting the line insertions, deletions, and modifications computed by the `diff` utility. However effective for comparing pairs of file versions, such tools cannot give an evolution overview of real-life projects that have thousands of files, each with hundreds of versions. Furthermore, they do not exploit the entire information potential of SCMs, such as information related to the time and author of changes between two versions.

Recent efforts try to overcome these shortcomings. The technique proposed by Collberg et al. [7] visualizes the software structure and mechanism evolution using a sequence of graphs. However, their approach does not seem to

scale well on real-life data sets. Gall et al. propose in [17] a 3D approach, using the third axis to display the evolution of structure in time. Their approach has also limited scalability and is subject to occlusion problems inherent to 3D representations. Lanza proposes in [24] a technique to visualize the evolution of object-oriented software systems at class level. Closely related, Wu et al. [47] visualize the evolution of entire projects at file level and visually emphasize the moments of evolution. Van Rysselberghe and Demeyer [40] propose a file based approach to visualize the change frequency of project artifacts. These methods scale very well with industry-size systems and provide comprehensive evolution overviews. Still, they focus on a high granularity level and do not address the lower-level system changes, such as the many, minute source code edits done during debugging. To cope with this limitation, Girba proposes in [19] a more generic approach to visualizing the history of any software entity.

We build on the above ideas and advocate the visualization of software evolution at different levels of detail and from different perspectives. Our approach is described next.

3. Data model

The examples presented in this paper contain data extracted from public CVS servers. CVS [9] is one of the most popular SCMs. Our approach and techniques are, nevertheless, generic, as we use concepts and structures that naturally exist in most SCM systems we are aware of. In this section, we describe our assumptions of these concepts and structures by means of a data model. The central element of a structure-based SCM system is a *repository* that stores all versions of a given file. A repository R is a finite set of files. Let NF be the number of files in R, then:

$$R = \{F_i | i = 1..NF\}.$$

Each file F_i is defined as a set of NV_i versions:

$$F_i = \{V_{ij} | j = 1..NV_i\}.$$

Each version is a tuple containing several *attributes*: the unique version id, the author who committed it to the repository, the time when it was committed, a log message and its source code:

$$V_{ij} = \langle id, author, date, message, code \rangle.$$

The first four elements of the above tuple (id, author, date and message) are unstructured attributes. A file version V can have any number of such attributes as the SCM system chooses to store, each attribute having some value. To keep the notation simple we drop the file index i in the following definitions. The fifth element (code) is modeled as a sequential structure, i.e. as a set of *entities*:

$$code = \{entity_i | i = 1..NE_i\}.$$

These entities can have different granularity levels, i.e. they can be code lines, scopes, functions, classes, namespaces, or even entire files, i.e. there can be different, alternative code data models. We make no assumptions whatsoever on how the versions are internally stored on the server. Our approach to modeling the evolution of this type of data is straightforward and maintains a generic form that makes it suitable for instantiation at different levels of detail.

Clearly, to visualize evolution, we need a way to measure change. We say two versions V_i and V_j of the same file differ if any element of their tuples differs from the corresponding element. Finding differences in the id, author, date, and message attributes is trivial. For the code attribute, we must compare the source code $code(V_j)$ and $code(V_{j+1})$ of two consecutive versions V_j and V_{j+1}. For this, we consider both V_j and V_{j+1} at the same granularity level, as it makes little sense to compare e.g. classes with code lines. If we choose the code line granularity level, we can use for comparison a tool like UNIX's `diff`, which reports the inserted and deleted entities in V_{j+1} with respect to V_j. All entities not deleted or inserted in V_{j+1} are defined as constant (not modified). Finally, entities reported as both deleted *and* inserted in some version are defined as modified (edited). We denote by l_i the ith entity of the version we talk about in some given context. Using `diff`, we can also find which entities in V_{j+1} match constant (or modified) entities in V_j. For each constant or modified entity l_i, we call the complete set of matching occurrences in all versions, i.e. the transitive closure of the above introduced match relation, the *global* entity associated with l_i, and we reference it by $L(l_i)$. Note that this technique can be applied at any granularity level, as long as we avail ourselves of a `diff` operator for the entity types of that level. Indeed, in Section 6, we shall demonstrate it for the component granularity level.

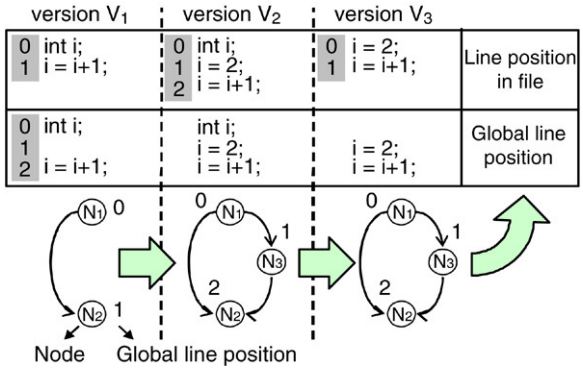

Fig. 1. Global line position and corresponding graph analogy.

From these data, we next build several functional characterizations of the source code evolution. The most important is the *global entity position*:

$$G(j, l_i) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}.$$

The purpose of this characterization is to assign to each entity a unique global position identifier that respects the local (i.e. inside one version) order relation. We can explain $G(j, l_i)$ by a graph analogy as follows. For every global entity $L(l)$, we build a graph node $N(L(l))$. Nodes are created by scanning versions V_j in increasing order of j, and entities l_i in each version in increasing order of i. For each consecutive pair of entities l_i and l_{i+1} in one version, we set a directed arc from $N(L(l_i))$ to $N(L(l_{i+1}))$. Finally, when a node N is inserted between two other nodes N_A and N_B, we set an arc from any already existing node between N_A and N_B to N. Fig. 1 shows three versions of a file with lines as entities. At the bottom of the image, the corresponding global position graph and its evolution are depicted. This graph is directed and acyclic, and gives a total order relation between all code lines in all versions of a given file. The node corresponding to the global line $L(l)$ before which no other line existed during the whole project is the only one having only outgoing arcs.

We label this 'start node' (e.g. node N_1 in Fig. 1, bottom) with zero and all other nodes with the maximal path length (defined as number of arcs) to the start node, e.g. by doing a topological sort of the graph (see [8]). We obtain then, for every line l_i in every version V_j, that $G(j, l_i) = label(N(l))$, where $l = L(l_i)$. This gives a unique label to all code lines written during development, keeps the partial line orders implied by the different versions in the project, and ensures that lines in different versions identified by `diff` as instances of the same global line have the same label.

A second functional characterization is the *entity status*

$$S(j, i) : \mathbb{N} \times \mathbb{N} \to STATES$$

which characterizes the status of entity with global position i in version V_j. S is computed by comparing the current entity l_C at global position i in version V_j with entity l_P having the same global position i in version V_{j-1} (i.e. the previous version), and entity l_N having the same global position i in version V_{j+1} (i.e. the next version). The *STATES* set contains the values described in Table 1.

Further information can be extracted from the source code. Using an appropriate parser one can extract structural information at each granularity level. We use for example a fuzzy parser with a user customizable grammar to extract information about constructs such as blocks, comments, preprocessor macros, at line level. This produces the *construct* attribute

$$C(j, l_i) : \mathbb{N} \times \mathbb{N} \to Grammar$$

which describes, for every entity l_i in every version V_j, the grammar construct that entity belongs to. We will use this information to visualize the structure of a given version in Section 4.

Finally, the last functional characterization we introduce is the *file position*

$$P(i) : \{1..NF\} \to \{1..NF\}.$$

Table 1
Possible values of the *STATES* set elements

Value	Condition
constant	{ l_p exists in V_{j-1} and is identical with l_C }
modified	{l_P exists in V_{j-1} but differs from l_C, or l_N exists in V_{j+1} but differs from l_C }
deleted	{ l_P exists in V_{j-1} and l_C does not exist in V_j or $S(j-1,i) = deleted$ }
inserted	{ l_N exists in V_{j+1} and l_C does not exist in V_j or $S(j+1,i) = inserted$ }
modified by deletion	{ l_C is modified, and $S(j,i+1)$ =*deleted* OR *modified by insertion* } **Note:** This signals that modification occurred in conjunction with deletion, and therefore it may have a larger impact. That is, the modified entity might be a completely new one.
modified by insertion	{ l_N is modified, and $S(j,i+1)$ =*inserted* OR *modified by insertion* } **Note:** This signals that modification occurred in conjunction with an insertion, and therefore it may have a larger impact. That is, the modified entity might be a completely new one.

This gives the position of a file in its corresponding project. Various instances of this characterization may exist, each offering a specific type of insight into the evolution of the system. In the examples we present in this paper, we use as *file position* the results inferred from different continuous or discrete metrics such as the file creation time, or the change activity measure (see Section 7). This enables the user to easily make correlations between the project files based on the metric values and the other entity-based functional characterizations.

Multiple levels of detail can be modeled using the functional characterizations presented in this section, by assigning various granularity levels to the entities. The examples we present in this paper address two extreme cases: the lowest granularity level, i.e. the line level, and the highest, i.e. the file level. The same mechanisms apply, nevertheless, when visualizing the project evolution at scope, function, class, namespace, package, or some other desired level.

We next detail the techniques used to map the data model described in this section onto visual elements.

4. Visualization model

Our work builds on the assumption that developers are most comfortable with visualizations that present the code in the same spatial context in which they construct it [10]. Since software maintenance is mainly done at code level, we decided to use a code-based approach to visualize the software evolution. Our main concern was to allow users to easily perform investigations by minimizing the cognitive overhead of multiple representations for the same data. For this, all our visualization techniques use a single-screen dense-pixel display for visualizing evolution at any given level of detail. Moreover, we integrated our techniques as separate views in the same toolset, so that users can easily switch between tasks such as opening a new repository, browsing through it, and requesting a detailed code-level view. We have implemented the above as an integrated toolset for SCM data extraction, analysis and visualization. Currently, this tool set contains two applications: CVSscan and CVSgrab. CVSscan visualizes the evolution of code lines as entities across a small number of files (i.e. it offers a *file view* on software evolution). CVSgrab visualizes the evolution of files as entities, across entire software projects (i.e. it offers a *project view* on software evolution). In the following, we detail the visualization techniques and methods used in CVSscan and CVSgrab. In Sections 5–7, we use CVSscan and CVSgrab to validate the presented methods and techniques on real-life, industry-size software projects.

4.1. Dimensions

Similarly to previous code-based software representations [10,20,15] we represent every entity (e.g. code line) as a pixel line on the screen. We took the decision to use a 2D representation. Our need to visualize many attributes together may first suggest using a 3D view. However, we opted for 2D in order to have a simple and fast user interface, no occlusion problems and viewpoint choice problems, and a visual layout perceived as simple for code developers. The main questions we next had to answer were how to lay out the entity representations in a plane, and how to use color and other visual cues for encoding data attributes.

Our layout approach is different in one major respect from previous code-based layouts. We visualize on the same screen all versions that an entity has during its evolution (Fig. 2), instead of all entities in a project at a given time.

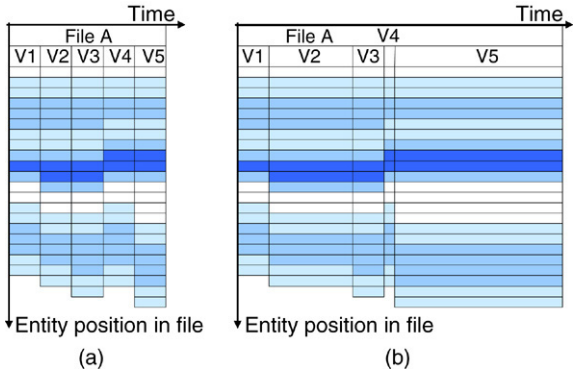

Fig. 2. Use of horizontal time axis in code-based visualizations (a) version-uniform time sampling (b) time-uniform time sampling.

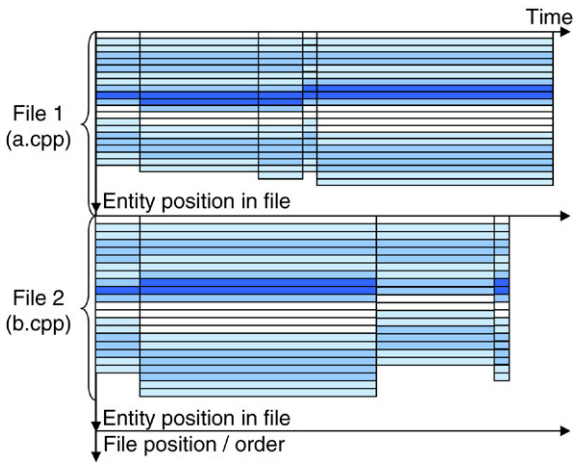

Fig. 3. Vertical entity/file position axis: entities are arranged according to their position in the file; files are arranged in alphabetical order.

The horizontal axis encodes the time. The vertical one the entity position l_i and the position of the containing file in the project.

Two main sampling strategies are possible on the time axis: version-uniform sampling and time-uniform sampling. In the version-uniform sampling, each version is shown as a vertical stripe composed of horizontal pixel bars depicting entities (Fig. 2(a)). This generates uniform incremental views on the evolution that are more compact and offer the same resolution both for 'punctuated' evolution moments, i.e. sharp variations of project size denoting important changes [47], and for equilibrium periods. This kind of sampling is more efficient for a set of entities that have many common change moments, such as all entities that belong to the same file, but it works as well for entities that belong to different files. For time-uniform sampling, each entity appears as a horizontal stripe, segmented according to its change moments (Fig. 2(b)). This generates views that are more suitable for placing the project evolution in the overall context, with punctuated and equilibrium periods, but makes correlation more difficult in the punctuated evolution area, due to lack of visual resolution. Both samplings are illustrated by examples in Figs. 5 and 8.

The vertical axis has a double role. First, it gives locally the position of an entity inside the containing file (e.g. the *local entity position*). Second, it gives globally the position of the file in the project it belongs to (i.e. *file position*) when the contents of more files are displayed in one image. Fig. 3 depicts an example of usage for the vertical axis for two files `a.cpp` and `b.cpp` in combination with a time-uniform sampling on the horizontal axis. The entities are arranged according to their position in the containing file. The files are arranged in alphabetical order. For a concrete example, see Fig. 8.

For the vertical layout of entities within one file strip, we propose two approaches. The first one, called *file-based layout*, uses as y coordinate the local entity position l_i (Fig. 4(a)). This layout offers an intuitive 'classical' view on file organization and size evolution, similar to [10]. The second approach, called *entity-based layout*, uses as y coordinate

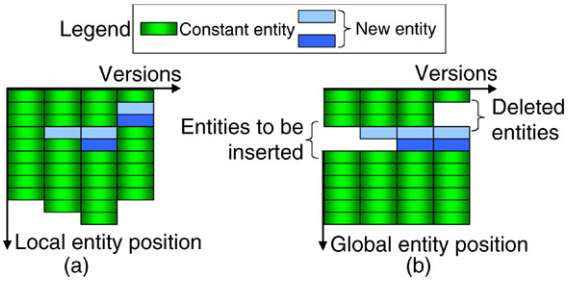

Fig. 4. Entity layout with a version-uniform time sampling: (a) file-based (b) entity-based.

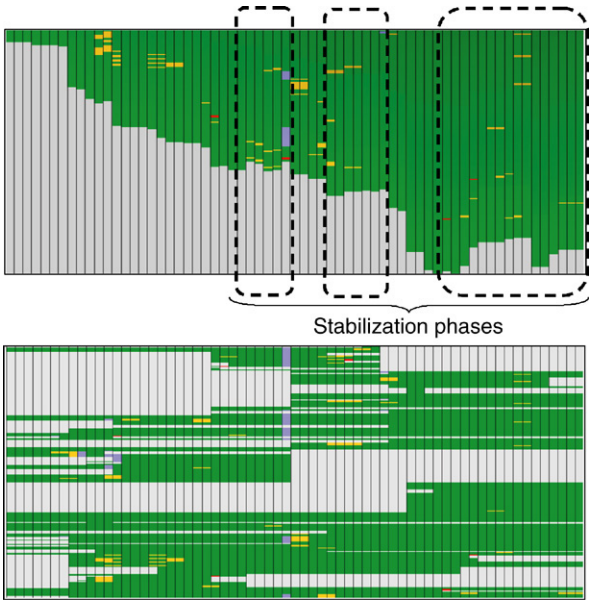

Fig. 5. Line status visualization with a version-uniform time sampling. File-based (top) and line (i.e. entity)-based layouts (bottom). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the global entity position $G(j, l_i)$ (Fig. 4(b)). While this preserves the order of entity of the same version, it introduces empty spaces where entities have been previously deleted or will be inserted in a future version. In this layout, each global entity has a fixed y position throughout the whole visualization. This allows easy identification of software entities that stay constant in time, or get inserted or deleted. Both approaches are illustrated by real data in Fig. 5.

To show various attributes, different color encodings may be used for the entity *status*, *construct* and *author* functional characterizations of a version. We use a customizable color map to indicate the status of entities in a given version. For the *construct* attribute (e.g. blocks, comments) we use a customizable color map, and modulate luminance to encode the nesting level. Finally, we use a fixed set of perceptually different colors to encode the entity authors. At each moment, one color scheme is active, such that the user can study the time evolution of its corresponding data attribute. When interesting patterns are spotted, one can switch to another scheme to get more detailed insight into the matter.

Next we present two examples of concrete implementations for the techniques we propose: the *file view*, which describes code changes at line level across multiple versions of a single file (or a small number of files), and the *project view*, which shows changes at file level across complete software projects.

File view

Fig. 5 shows the visualization of a file evolution through 65 versions using CVSscan. CVSscan provides a *file view* on the software evolution. It uses the code line as level of detail for the entity granularity, and a version-uniform sampling of the time axis. In this example color encodes *line* (i.e. entity) status: green denotes constant, yellow modified, red modified by deletion, and light blue modified by insertion respectively. Additionally, in the line-based layout (bottom),

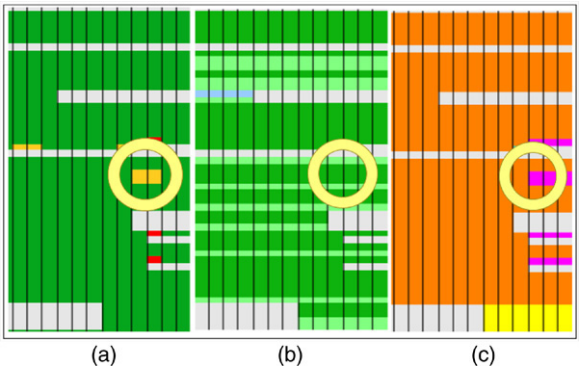

Fig. 6. Attribute encoding: (a) line status; (b) construct; (c) author. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

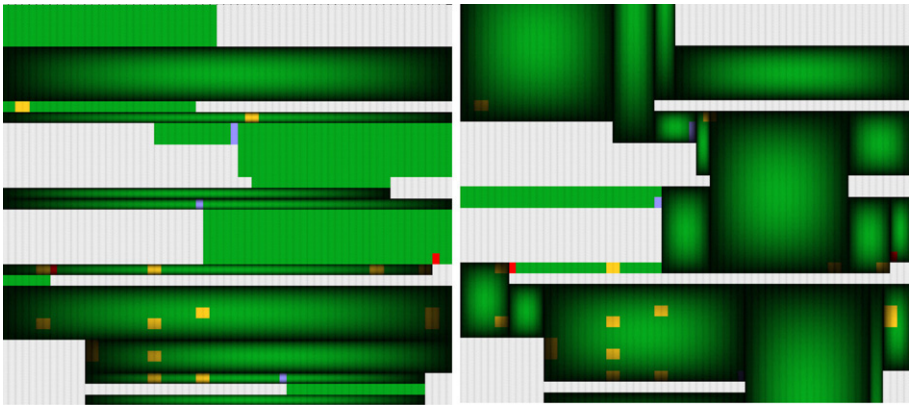

Fig. 7. Detection of stable code fragments (a) version-based (b) line-based. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

light gray shows inserted and deleted lines. The file-based layout (top) clearly shows the file size evolution and allows spotting of the stabilization phases occurring during the project. That is, phases when the file size has a small decrease corresponding to code cleanup, followed by a relatively constant size evolution corresponding to testing and debugging. Yellow fragments correspond to areas that need reworking during the debugging phase.

Fig. 6 illustrates different color encodings on a zoom-in of the line-based layout in Fig. 5 (bottom). In Fig. 6(a), we use yellow to encode lines that suffer modifications when passing from one version to another, as shown in the highlight. Since the *modified* state is by definition symmetric (see Section 3), yellow lines always appear in pairs. Switching to the color scheme that encodes the *construct* attribute (Fig. 6(b)) enables the user to discover that the modified piece of code is in a comment, encoded by the dark green color. This means the modification does not actually alter the code functionality. Finally, the *author* attribute (Fig. 6(c)) shows the developer that performed the modification, (e.g. the purple one in our highlight).

The file view perspective can be also used to visualize stable code fragments, i.e. code containing no insertions or deletions, without modifying their layout and/or their attribute encoding color schemes. Two methods for stable code blocks are provided: version-based and line-based. Both methods detect contiguous intervals in the version-line (x–y) space that contains no deletions and/or insertions and deliver a list of such intervals, sorted as follows. The line-based method maximizes first the number of lines (x axis) and is useful for revealing the long code fragments that are stable for a certain evolution time. The version-based method tries to maximize first the number of versions (y axis) and is useful for revealing the code blocks that are stable for long periods. To display the code blocks, shaded cushions are used [41]. Size thresholds are used to interactively limit the search, e.g. to answer queries like "show all stable blocks longer than X lines or that have existed for more than Y versions". Fig. 7 shows the two methods on a code fragment of 50 lines followed along 65 versions. Color maps the line status attribute: dark green =

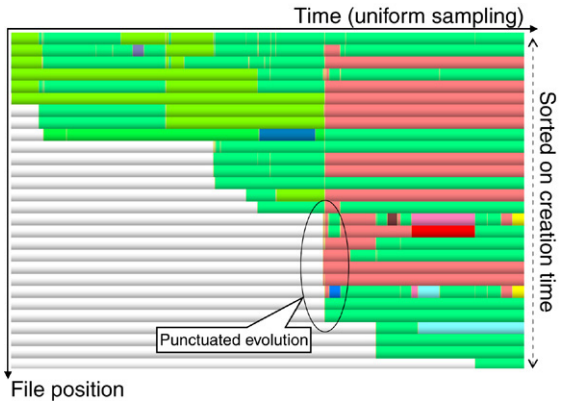

Fig. 8. File author visualization with a time-uniform sampling and files arranged in order of creation time.

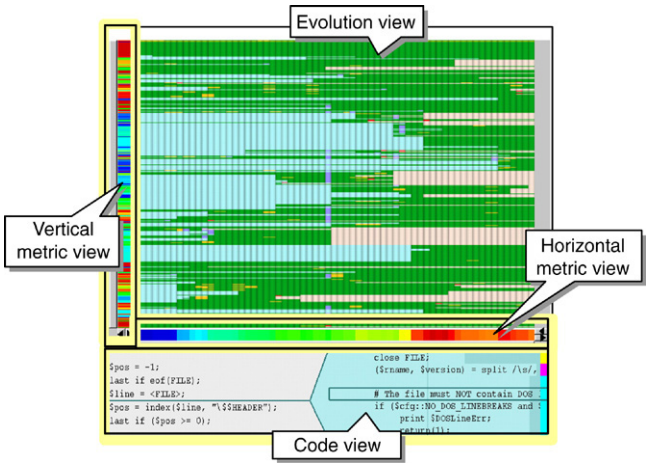

Fig. 9. Multiple code views in CVSscan.

constant, yellow = modified, light gray = inserted or deleted, red = modified by deletion, light blue = modified by insertion.

Fig. 7(a) shows several wide blocks that have been stable for almost the entire evolution. Fig. 7(b) shows the longest (tallest) stable fragments. Color encodes line status, as explained above — the small yellow blocks visible in the lower part of the images indicate, for example, that only a few lines of code were edited. However, the large empty spaces in the layout indicate that many lines were inserted and deleted.

Project view

Fig. 8 uses CVSgrab to visualize the evolution of a project containing 28 files across up to 20 versions spanning 4 development years. CVSgrab provides a *project view* on the software evolution. It uses the file as level of detail for the entities and a time-uniform sampling of the horizontal axis. In the depicted example color encodes the author responsible for a given commit — different colors show different authors. Files are arranged in order of creation time, from top to bottom. One can easily identify in Fig. 8 the main contributors and the punctuated evolution moment that occurs around the middle of the development period.

4.2. Metric views

A key factor in understanding the patterns revealed by evolution visualization is to correlate them with other information about the software. Besides the code-based visualization of code evolution we presented so far, we propose using additional metric views and a novel text view on selected code fragments (Fig. 9).

The metric views encode time and entity-dependent data and show these with vertical, respectively horizontal, color bars to complement the evolution visualization. Different metrics are available. In Fig. 10, we show three

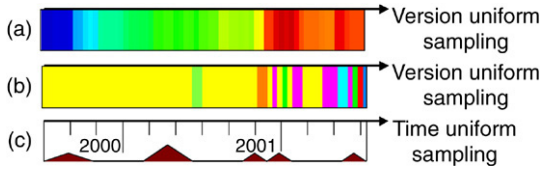

Fig. 10. Horizontal metric bars: (a) version size; (b) version author; (c) activity density. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

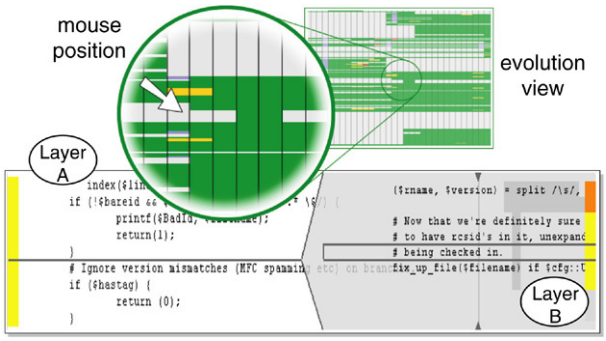

Fig. 11. Two-layered code view correlated with a version-uniform sampling entity layout.

proposed horizontal metric bars that illustrate, for each version, its number of entities or its author for a version-uniform sampling on the time axis, and the activity density (i.e. number of modifications on a specific time interval) for a time-uniform sampling. The top-most metric (version size) uses a classical blue-to-red colormap, where blue indicates low values, green average ones, and red high values. The middle metric (version author) uses a simple hue-to-author encoding: every author gets a distinct hue from a predefined palette. The lowermost metric (activity density) uses a simple graph plot where the graph height indicates the metric value.

Similarly, the vertical metric bars can display various metrics computed on the entire evolution of an entity for a given global entity position, e.g. the entity life time, the entity similarity to a reference entity, the entity activity intensity, and so on. Concrete examples of using metric bars are given throughout the paper (e.g. Figs. 13, 14 and 22).

4.3. Code view

The code view offers a detailed text look at the actual source code. Users can select the code to be displayed by sweeping the mouse in the evolution view. Vertical brushing, i.e. moving the mouse over, the code evolution area scrolls through the program code at a specific moment, whereas horizontal brushing, for example over the entity-based layout (Section 4.1), goes through a given entity's evolution in time.

An important issue we address in our work is how to correlate the code and evolution views, when the latter uses an entity-based layout. The question is what to display when the user brushes over an empty space in the evolution view. This space corresponds to *deleted* or *inserted* entity status values, i.e. the code at the mouse position was deleted in a previous version or will be inserted in a future version (e.g. the light gray areas in Fig. 5). Freezing the code display would create a sensation of scrolling disruption, as the mouse moves but the text doesn't change. Displaying code from a different version than the one specified by the mouse position would be wrong.

We solve this problem by a new type of code display. We use two text layers to display the code around the brushed global entity position both from the version under the mouse and from versions in which this position does not refer to an empty space (Fig. 11).

While the first layer (A) freezes when the user brushes over an empty region in the evolution view, the second layer (B) pops-up, and scrolls through the code that has been deleted, or will be later inserted at the mouse location. This creates a smooth feeling of scrolling continuity during browsing. At the same time, it preserves the context of the selected version (layer A) and gives also a detailed, text-level peek at the code evolution (layer B). The three motions (mouse, layer A scroll, layer B scroll) are shown also by the captions 1, 2, and 3 in Fig. 13.

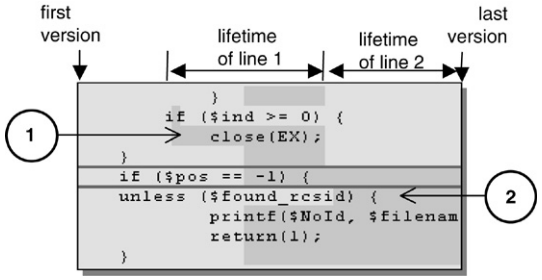

Fig. 12. Code view, layer B. Line 1 is deleted before line 2 appears, i.e. they do not coexist.

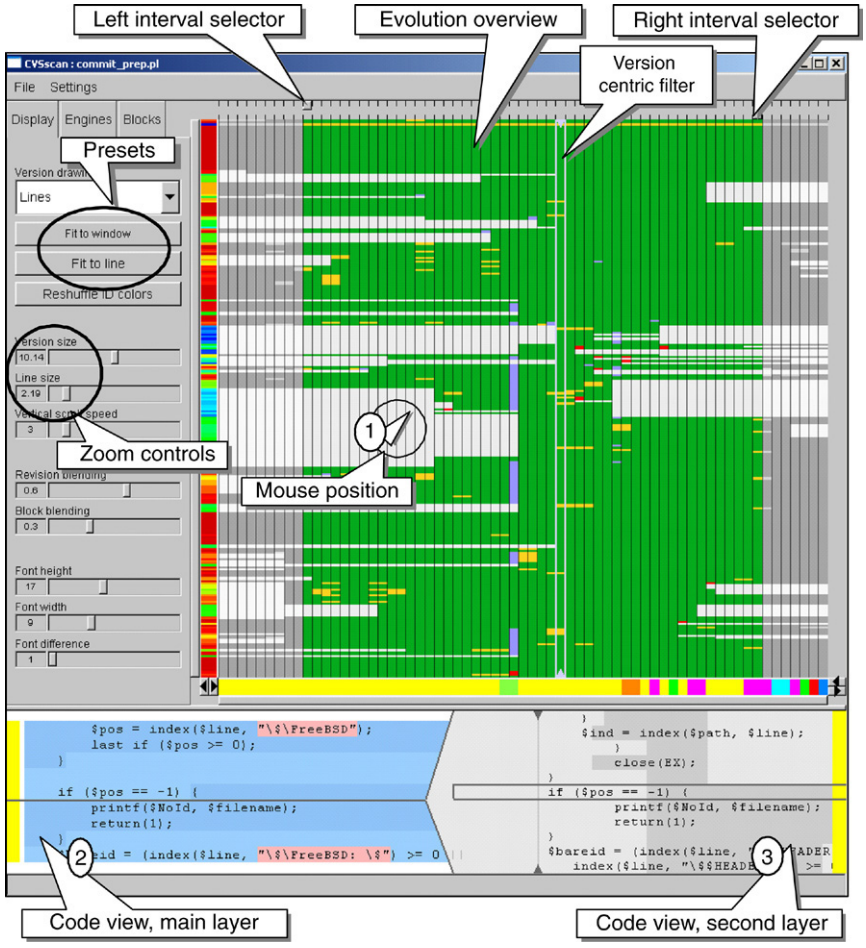

Fig. 13. CVSscan tool overview. The file version and line number under the mouse (1) is shown in detail in the text views (2, 3).

We must now consider how to assess the code evolution shown by layer B. The problem is that lines of code corresponding to consecutive global entity positions might not coexist in the same version. In other words, layer B consecutively displays code lines that may not belong to one single version. We need a way to correlate this code with the evolution view. We achieve this by showing the entities' lifetimes as dark background areas in layer B (Fig. 12). Finally, we indicate the author of each line by colored bars along the vertical borders of the code view (Fig. 11).

Summarizing, the code view offers a detailed look on a specific global entity position in a selected version, including information about its evolution and the developers that make it happen. We use this view in CVSscan to correlate the line evolution with detailed line information.

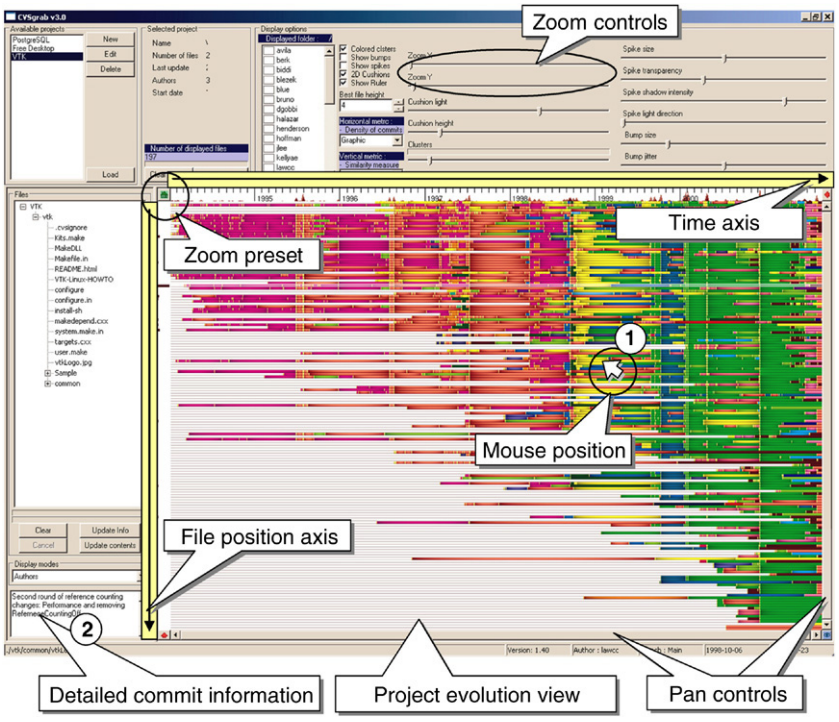

Fig. 14. CVSgrab tool overview. The detailed comments of the version selected by the mouse (1) are displayed in a correlated view (2).

4.4. User interaction

In addition to the described visualization techniques, our toolset offers a wide range of interaction means to facilitate the navigation of data. We describe below, using the well-accepted perspective proposed by Shneiderman [34], the palette of interactive exploration instruments we provide. All instruments use a point-and-click approach, making the entire exploration very simple to use. Tool snapshots illustrating these mechanisms are shown in Figs. 13 and 14.

Our toolset offers an intuitive **overview** on the evolution of software in a single 2D image, even when the number of entities is larger than the available screen resolution of the monitor. To get more detailed insight in a specific region of the evolution, **zoom** and **panning** facilities are provided. This enables the user to drill down to more detailed representations, in which the evolution of each entity/file may be assessed. The tools offers also preset zoom levels: global overview (fit all code to window size) and one entity-per-pixel-line level.

In order to support the file evolution analysis from the perspective of one given version, CVSscan offers a **filtering** mechanism by means of which all lines that are not relevant are removed from the visualization (i.e. lines that will be inserted after the selected version, or lines that have been deleted before the selected version). Filtering enables the user to assess a version, selected by clicking on it, by clearly identifying its lines that are not useful and will be eventually deleted, and the lines that have been inserted into it since the beginning of the project. In other words, filtering provides a version-centric visualization of code evolution. Additionally, the tool gives the possibility to **extract** and select only a desired interval to study the program evolution. This mechanism is controlled by two sliders (Fig. 13, top) similar to the page margin selectors in word processors. By choosing the starting and finishing version, one can remove from visualization the code that is not relevant (i.e. code deleted before the starting version, or code inserted after the finishing one). This mechanism proved to be useful in projects with a long lifetime (e.g. over 50 versions) in which one usually identifies distinct evolution phases that should be analyzed separately. The distinct phases were identified using a *project view* similar to the one presented in Fig. 8, after which detailed *file views* were opened and the time span of interest was selected using the version sliders described above.

Both CVSscan and CVSgrab enable the user to **correlate** information about the software evolution with overall statistic information (by means of the metric views) and with specific details. By means of metric bars, users can

visually get statistical information about lines (e.g. the lifetime of a line at a given global position), or versions (e.g. a version's author or size). The bi-level code view (Section 4.3; Fig. 13, captions 2 and 3) offers **details-on-demand** in CVSscan about a code fragment: the text body, the line authors and the text evolution. The user can select the fragment of interest by simply brushing the file evolution area. Similarly, in CVSgrab, the user can obtain detailed information about the brushed version in the form of user commit comments (Fig. 14, caption 2).

Although CVSscan and CVSgrab are purely exploration tools that do not alter the visualized data, they maintain a collection of state variables that can be externalized. This enables the user to keep a **history** of his actions and let him recover and reuse specific visualization settings at a later time. In this direction, a simple extension our users suggested was to add an annotation facility by which they can add their own comments, and visualize added comments, to a given time and entity position. This enables a quick and effective way to store insights gained during visualization sessions.

We next present the results of three informal studies performed using the CVSscan and CVSgrab tools. These studies show how the interaction mechanisms and the visualization techniques described so far can be efficiently used to investigate the evolution of real-life software systems.

5. User study 1: Evolution of files

The main target audience of the techniques we describe in this paper is the maintenance community. The maintainers perform their tasks outside the primary development context of a project, and most often long after the initial development ended. Therefore, the main activities a maintainer performs are related to context recovery, such as program understanding and team network building. CVSscan facilitates this process by visualizing file evolution from the perspective of different attributes and features, such as file structure, modifications, and authors.

In order to validate the visualization techniques and methods in CVSscan, we organized a number of informal studies. The aim was to record and analyze the experiences of software maintainers when they investigate completely new programs, i.e. programs in whose development they did not participate, with no other support than CVSscan itself. The task to be accomplished was to quickly get familiar with the overall code and give a description of several important changes that took place during the evolution. We present below the outcome of two such studies of the larger set we organized. In both cases, the users had not previous experienced with CVSscan. Therefore they first participated in a 15 min training session. During the session, the tool's functionality was demonstrated on a particular example file. After that, each user was given a file for analysis, but no information about its contents whatsoever. A silent observer recorded both user actions and findings.

5.1. Case study 1: Analysis of a Perl script file

In the first case, the user was given a script file from the FreeBSD distribution of Linux, containing 457 global line positions and spanning 65 versions. The user had programming knowledge and was familiar with scripting languages, but had no advanced knowledge about Perl. The user started CVSscan using the default file-based layout to visualize the evolution of file structure. The navigation steps the user took during the study are sketched in Fig. 15 with half-transparent arrows.

The user brushed first over the green areas in the evolution view: "These are comments... Let's see first what they say".

He started to brush from the beginning of the file, choosing first the comments that spanned over the entire evolution. In the same time he read the code fragments displayed in the code view. "This is Perl. All Perl scripts have this path on the first line. This one looks like a file description. It reads that this script handles pre commits of files...".

Then while brushing over the comment fragments (Fig. 15(a) top → bottom): "These are annotated textual dividers: *Configurable options*, *Constants*, *Error messages*, *Subroutines*, *Main body*. I use these too in my programs... Here are also some annotations...".

Further on, the user investigated also the large comment fragments that did not span over the whole evolution: "It looks like the implementation was either not completed or the developers left a lot of garbage. There are some code fragments over here that are commented out".

The user next selected the last version and brushed over the *Subroutines* area "It looks like these lines do not belong to any block. Here is a blank line before the `write_line` procedure. Here a blank line before `exclude_file`. So there

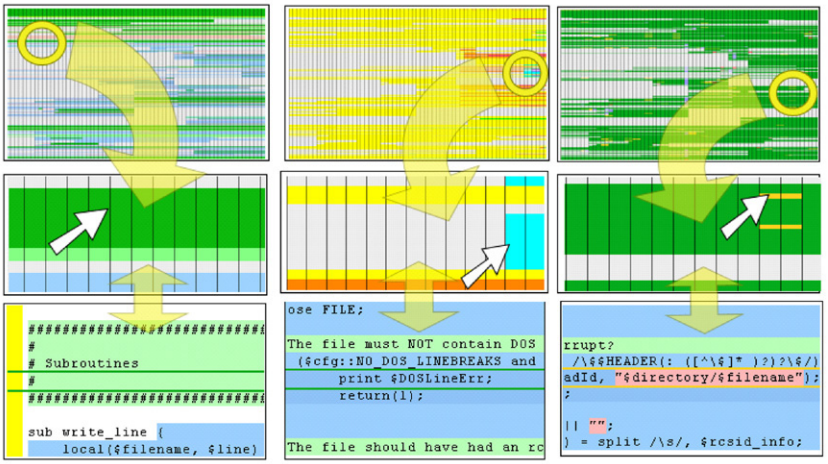

Fig. 15. Case study 1 — Analysis of a Perl script. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

are white lines before every procedure? Yes, indeed: `check_version`, `fix_up_file`. So there are four procedures. It seems `exclude_file` is the most complex one as it has the highest nesting level".

At this point, the user had a high-level understanding of the file structure. He started to make inquiries about the developers that had worked on the file. For that, he switched back and forth between the *construct* and *author* attributes using shortcut buttons: "The yellow developer, Dawes, did most of the work. However, the orange one, Robin, wrote that complex `exclude_file` procedure. He did that towards the end of the project, so probably that adds some extra functionality to the core. I see also that the cyan developer, Eich, did some significant work towards the end in the `check_version` procedure (Fig. 15(b) top → bottom). It seems that his concern was to rule out files containing DOS line breaks... So this script doesn't handle DOS files?".

The user then dismissed the authors that had only small contributions and switched to the *line status* visualization: "Apparently a major change took place in the middle of the project. It mainly affected the `check_version` procedure".

Then, selecting the version that followed the *modified by insertion* lines of the major change, the user started to concentrate on the areas where modifications took place: "I see a number of modifications between these two versions (Fig. 15(c) top → bottom). The first one replaces a file reference with a fully qualified name; the second does the same, the third too, the fourth, the fifth. Oh, they should have kept that file name in a separate variable!". "Here they tuned the regular expressions". "Here they replaced a constant string with a variable".

The user continued to brush all areas where modifications appeared and tried to correlate them with the code and the authors that committed them. We interrupted the experiment after 15 min. At the end of the exercise, the user was familiar with the overall organization of the file, the focus of each individual contributor, the places that had gone through important modifications and what these modifications referred to.

5.2. *Case study 2: Analysis of a C code file*

In the second case, an experienced C developer was asked to analyze a file containing the socket implementation of the X Transport Service Layer in the FreeBSD distribution of Linux. The file had 2900 global line positions and spanned across 60 versions. The user was not involved in any way in the development of, or familiar with, the examined software. We provided the user with a CVSscan version able to highlight C grammar and preprocessor constructs (Section 3), such as `\#define`, `\#ifndef`, etc.

The user started the tool in the default mode too, and tried first to look for commented fragments: "This is the copyright header, pretty standard. It says this is the implementation of the X Transport protocol, pretty heavy stuff... It seems they explain in these comments the implementation procedure...".

The user next switched his attention to the compiler directives: "A lot of compiler directives. Quite complex code, this is supposed to be portable on a lot of platforms. Oh, even Windows".

Next, the user started to evaluate the inserted and deleted blocks: "This file was clearly not written from scratch, most of its contents has been in there since the first version. Must be some legacy code... I see major additions done in

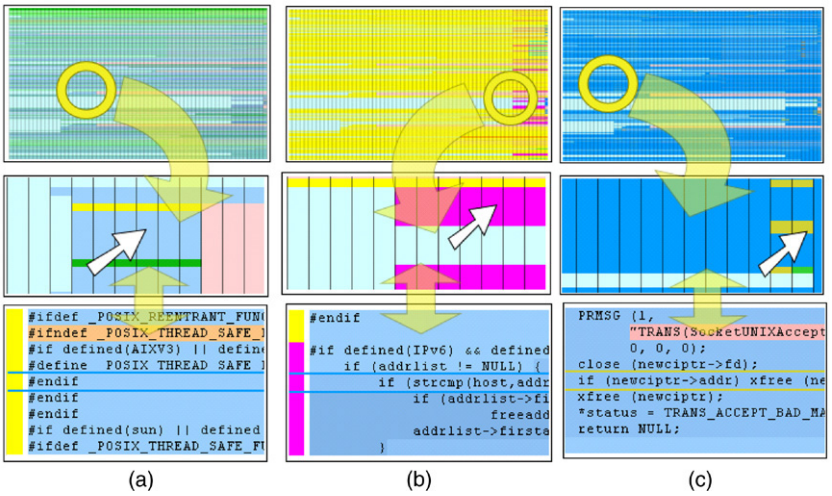

Fig. 16. Case study 2 — Analysis of a C code file. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the beginning of the project that have been removed soon after that... They tried to alter some function calls for Posix thread safe functions (Fig. 16(a) top → bottom)... I see major additions also towards the end of the project... A high nesting level, could be something complex... It looks like code required supporting IPv6. I wonder who did that?".

The user switched then to the *author* visualization: "It seems the purple user, Tsi, did that (Fig. 16(b) top → bottom). But a large part of his code was replaced in the final version by... Daniel. This guy committed a lot in the final version... And everything seems to be required to support Ipv6. The green user, Eich, had some contribution too... well, he mainly prints error messages".

Eventually, the user switched to the evolution of *line status* and used the predefined "Fit to line" setting to zoom in. "Indeed, most work was done at the end... Still, I see some major changes in the beginning throughout the file... Ah, they changed the memory manager. They stepped to one specific to the X environment I assume. All memory management calls are now preceded by x (Fig. 16(c) top → bottom)... And they have given up the `TRANS` macro".

The user spent the rest of the study assessing the modifications and the authors that committed them. We interrupted the study after 15 min. At the end, the user did not have a very clear image of the file's evolution. However, he concluded that the file represented a piece of legacy code adapted by mainly two users to support the IPv6 network protocol. He also pointed out a major modification: the change of the memory manager. All findings were checked by subsequent detailed code analysis.

5.3. Study conclusions

Although informal, the organized studies showed that the line-based evolution visualization of code supports a quick assessment of the important activities and artifacts produced during development, up to the source code line level of detail, even for users that had not taken part in any way in developing the examined code. Additionally, the approach is relatively generic as it may be applied to study the evolution of any line-based structure. The user-study subjects valued most the compact overview coupled with easy access to source code. These enabled them to easily spot issues at a high level and then get detailed source code information.

During the organized informal studies CVSscan was applied on evolution of files ranging up to 2900 lines of code and spanning along up to 120 versions covering more then 10 development years (e.g. files from the VTK library project). For additional studies see [25]. The tool scaled well in such cases, thanks to its embedded position-based antialiasing algorithm [42] and its convenient navigation features, e.g. the preset based zoom.

A weak point of the CVSscan tool at this moment is the accuracy of the `diff` operator used to discover differences between versions of the same file. CVSscan uses the `diff` operator provided by CVS repositories. The visualization accuracy is relative to the heuristics behind this operator, which can lead to data misinterpretations. A significant improvement would be to use a `diff` tool with support for semantic comparisons.

6. User study 2: Evolution of components

Component based software engineering is regarded as a promising approach towards reducing software development time and costs. While it has proven to be successful in many application domains such as office and distributed internet-based applications, the component-based approach toward development is still to be validated in the area of dependable systems, which have special requirements on quality attributes. Mller et al. [29] have elaborated a set of such requirements, and classified a number of existing component models according to their conformance to the set. However, as the number of component models increases, a new challenge arises: how to discriminate among models that satisfy the same set of requirements such that the best suited one is selected as the development base for a given system. Using the evaluation methodology proposed in [5], one can easily reach the conclusion that for example the Koala [31] and PECOS [46] component models offer similar benefits from the point of view of testability, resource utilization, and availability of a computational model. When these are the only important nonfunctional requirements for the component framework, the selection of the best suited model can be further refined with information on which model fits better with the software development process that will be further used during the project's lifecycle.

We next show how CVSscan was used to asses the development process associated with a given component model, by studying history recordings about the component structure evolution. We present two mechanisms for distinguishing among component models based on information about component structure evolution, as follows: assessment of the component development process (Section 6.3); and assessment of change propagation from framework to components (Section 6.4). We illustrate the above techniques with examples from the ROBOCOP component framework [22].

6.1. Component system assessment

Given the data model and visual mappings supported by our toolset (Sections 3–4), the question arises of how to use them to assess component-based software. We are interested in assessing the component development process for a given component model. To use our toolset, or similar ones such as its predecessor SeeSoft [10], for component-based source code, several assumptions implied by the toolset's data model must hold, as described in Section 3. First, the code should be describable by a sequential entity structure. In our current case, these are the components. Secondly, information about entity changes should be available, i.e. we should be able to say when a component was modified, inserted, or deleted.

Assessing a component development process is, in this respect, a conceptually simple task. Indeed, virtually all component models have a clear mapping between components and their source code. Additionally, many software projects use software configuration management systems, similar to CVS, which hold history records about code evolution and offer mechanisms for code comparison.

We next present three mechanisms for assessing a component model based on visualizations built with the CVSscan tool. We illustrate these mechanisms with real life examples. All these examples have been obtained from the ROBOCOP framework for component-based software development [22].

6.2. Context of assessment

To understand our results, we first sketch the use context. ROBOCOP was developed over a period of four years by an international consortium of several industry and academic partners. Its focus is on providing a generic, flexible, and resource-efficient set of mechanisms and tools for implementing, composing, deploying, and monitoring component-based software applications with a focus on high volume embedded appliances such as mobile phones, set-top boxes, and embedded controllers. In this respect, the ROBOCOP component model is similar to Koala [31], Rubus [1] and PECOS [46]. ROBOCOP's core, the run-time environment (RRE), acts as a virtual machine providing application-specific component library management, component instantiation and destruction, and inter-component communication and control interfaces. Component interfaces are described in the ROBOCOP Interface Description Language (RIDL). Just as in most component frameworks, this language can be translated to generate classical C skeleton code. Developers can next add component implementation code to this skeleton and compile it on a variety of platforms. For both the RRE and the component libraries, several tens of versions exist, developed by the different consortium partners during its four-year history. These versions emerged either due to changes in the framework and/or

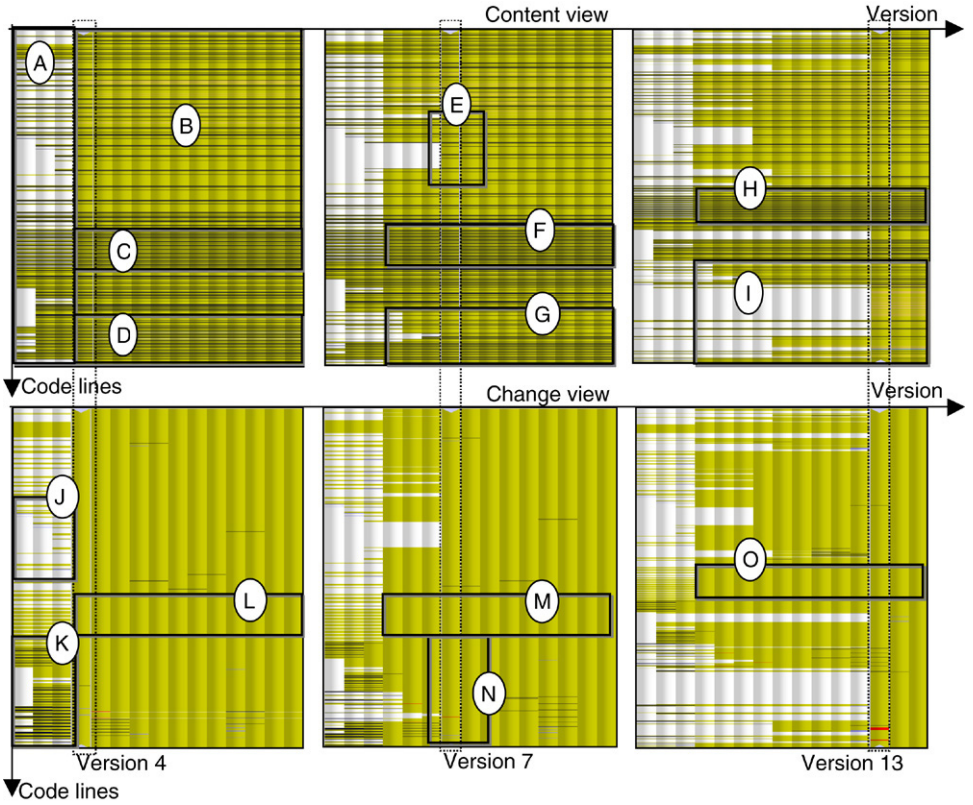

Fig. 17. Visualization of ROBOCOP component development process. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

component interfaces or due to implementation changes when porting these to different hardware platforms. Finally, we mention that CVSscan was used in assessing the ROBOCOP framework by different people than the component framework developers. We had thus the added difficulty of understanding a third-party large software system mainly through the perspective of our CVSscan visualization tool.

6.3. Assessment of component development

The choice of the component development process is an important issue when selecting the base component model on which a software system should be built. Investigating history recordings can give an indication about the effort required to have a minimal working component for testing purposes, the effort for modifying component interfaces and/or implementations, and the overhead related to maintaining framework compliance during development. Questions such as "how hard is it to build a minimal component?" or "how much component code was changed when some framework interface got added?" can be answered by CVSscan. Once we have been able to answer several such questions concerning the *past* evolution of our system, we attempt to extrapolate the results to the future. Most partners of the ROBOCOP consortium have expressed their high interest in being able to answer, even if only qualitatively, the above questions.

Fig. 17 depicts several CVSscan visualizations of *one* real-life ROBOCOP component evolution along 15 versions. We use a line-based layout, so the evolution of each source code line can be easily followed along the horizontal time axis. The upper part (*Content view*) shows three snapshots of the chosen component evolution from the perspective of three different versions: version 4, version 7 and version 13. Color encodes line content, extracted by our custom-grammar based parsing (Section 3): black shows function headers, yellow shows function bodies. The three snapshots in the *Change view* part of Fig. 17 correspond to the ones in the *Content view*, but use a different color encoding: black indicates lines that change from one version to the next, while yellow encodes constant (i.e. not modified) lines. In all snapshots, white shows gaps in the code, i.e. places where code was deleted in a previous version or will be

inserted in a future version, as described in Section 4. We can use Fig. 17 to understand our component development process. In the beginning (A), the developer tries to build a stable component interface. He edits the RIDL component description file and then generates a C source code skeleton using the RIDL compiler. In the first three versions of the considered use case, the developer does not add implementation code to the generated C skeleton, but tries to refine the RIDL interface description. This is apparent in Fig. 17 (J), as no code is inserted in the visualized file besides the C function headers automatically generated by the RIDL compiler, i.e. the thin dark lines inserted in versions 2 and 3. Additionally, we might also infer the fact the code is automatically generated. This is hinted by the function headers in region K, that change (i.e. are black) together with the insertions in version 2 and 3. While these might be voluntary changes, the modification of the same headers every time a new header is inserted signals the fact that code generation might not be entirely under the control of the developer. Indeed, in ROBOCOP, generated function headers have an automatically created textual reference to the line number in the RIDL description file that corresponds to that function. Inserting new interface specifications causes the textual references to the interface specifications following after them to change, since the specification location in the file changes.

As depicted in Fig. 17, this can lead to misunderstandings, due to the current skeleton generation process, as follows. Every time the developer adds new specifications to the RIDL file, he needs to run it through the RIDL compiler in order to generate appropriate function headers, which cannot be created by hand. However, once developers start to manually fill in this generated skeleton, adding new interfaces can be a very cumbersome process. This is mainly caused by the RIDL compiler, which cannot merge new skeleton information with existing ones, but generates the entire skeleton anew, discarding any hand-coded additions done by the developer. To prevent this, users maintain copies of the old code, and every time new interfaces are added, the generated function headers are manually merged by cut-and-paste in the saved copy. In this way, however, textual inconsistencies are introduced in the existing function headers as they reference invalid locations in the RIDL specification file. This can be seen also in Fig. 17, by comparing area (K) and (N). The introduction of new interfaces in versions 2 and 3 (J) causes existing function headers to be updated (K). However, in version 7, the developer manually inserts automatically generated headers in the previous file version (E), which causes no update in the existing headers (N).

Fig. 17 shows also the amount of effort required to have a minimal component running for testing purposes. Version 4 of the considered component was also the first functional one. We identify the main effort to achieve that as writing the code in the (B) area. From the *Change view*, we also see that the code required for a minimal component does not change in time except for some additional interface additions like the one highlighted in (E).

The evolution of 'useful', not automatically generated, component code can be seen in the areas D, G and I, where most of the code inserted during component refinement (versions 4 to 15) goes away. Areas C, F, and H in the *Content view* and the corresponding regions L, M, and O in the *Change view* refer to empty function stubs, i.e. unimplemented interfaces. These give the code overhead required for compliance with the ROBOCOP framework and have no other useful purpose for the component functionality.

Summarizing Fig. 17, we concluded that developing ROBOCOP components requires very careful code architecting. Subsequent interface changes are difficult to accommodate or lead to inconsistencies. Additionally, the effort required to have a minimal ROBOCOP component running is relatively high, accounting for almost 50% of the code in the presented example. However, once we have this code, it does not change significantly during further refinement of the component. Eventually, ROBOCOP components must always have several pure 'overhead' functions with empty implementations for compliance with the framework.

6.4. Assessment of change propagation

When component frameworks are not yet mature, it is often the case that new framework versions are not compatible with previous ones. In such cases, existing components need to be re-architected to various degrees in order to be supported by the new framework. The effort required for this step may be so high that migrating to a totally different, more mature, component framework or staying with the old framework may be better alternatives. A good estimation of the transition cost of framework change is therefore of great importance. CVSscan can help make such estimations, based on history recordings for components that have been already re-architected to comply with new framework versions.

Fig. 18 shows four CVSscan visualizations for the evolution of the same component as discussed in Section 6.3, but including two additional versions that correspond to the transition from a first framework version to a second one.

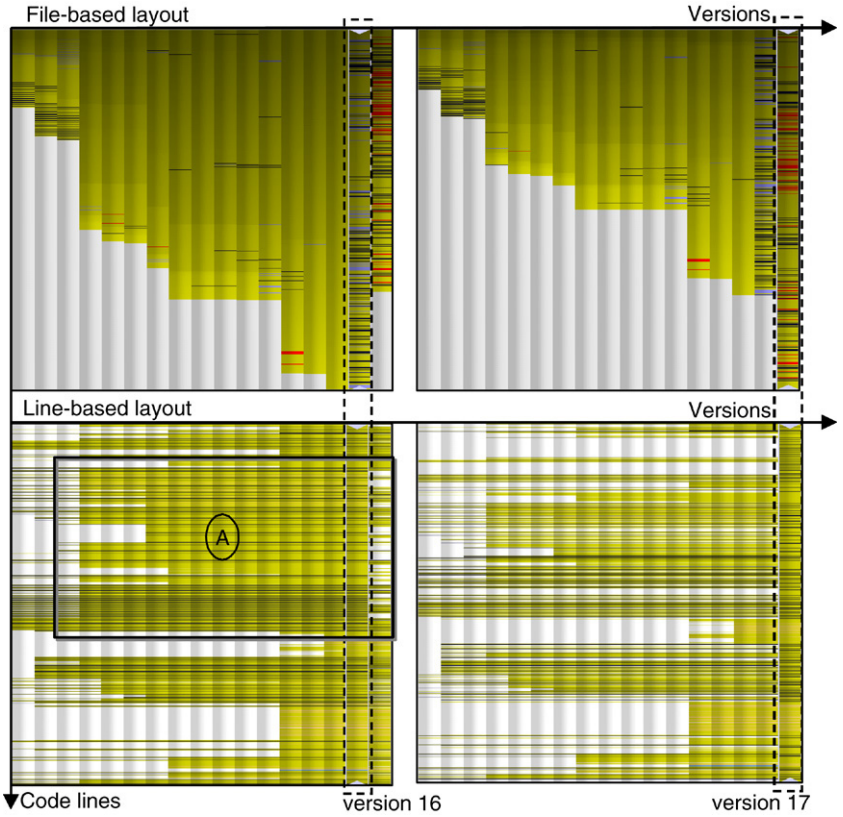

Fig. 18. Migrating a component from ROBOCOP 1.0 to ROBOCOP 2.0. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In Fig. 18 (top row), we use a file-based layout in conjunction with a version filter (see [42,43]) to depict the amount of code from one version that can be found in other versions. Color encodes change: yellow (light) areas are lines that did not change during development; black (dark) areas show line changes. From this image, we infer that a lot of code had to be changed when passing from component version 16 to version 17. Additionally, only about 75% of the component implementation code from version 16 is found in version 17. Furthermore, the top right image shows that about 40% new code had to be written for version 17 in addition to what was preserved from version 16. Overall, about 50% of the component code in version 17 differs from that in version 16. This signals a quite high effort to adapt components to cope with changes in the ROBOCOP framework.

Fig. 18 (bottom row) uses a line-based layout (Section 4.1) together with a version filter to show what interfaces have been removed and what was inserted during re-architecting. Color encodes line content: black (darker) = function headers; yellow (lighter) = function bodies; white = deleted or inserted code, just as in Fig. 17 (see Section 6.3). Correlating the lower left image (A) with the content evolution images from Fig. 17, we easily see that a major benefit of migrating to the new version 2.0 of the ROBOCOP framework was to decrease the number of mandatory interfaces that a component must implement to be compliant with the framework.

CVSscan allows also more in-depth analysis of the re-factoring a component passes through. This allows us, when browsing the code, to separate framework-induced code changes from those attempting to achieve a better design for the component itself. Fig. 19 depicts such a case. We use here a line-based mapping to show the evolution of a component's code over 10 versions. The same color scheme as in Figs. 17 and 18 is used to display line changes. In Fig. 19, we can quickly see an abrupt change performed in version 8. At first glance, we believe we see the addition of several component interfaces (red arrow highlight, A) and deletion of some existing ones (blue arrows highlight, D). However, a closer analysis of the image (B) shows that all function declarations from version 7 are also found in version 8, and the actual code deletions refer only to parts of the implementation (function bodies). Moreover, the newly introduced functions (A) are not interface implementations. Using the code view (Section 4.3), we can easily

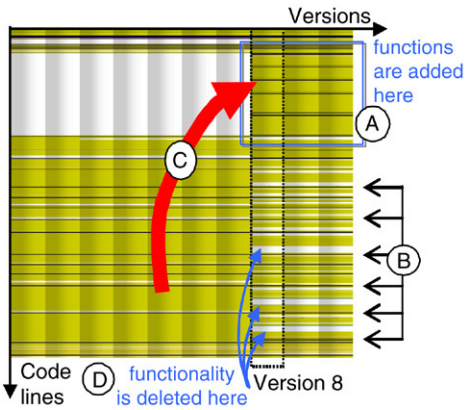

Fig. 19. ROBOCOP component re-factoring: factorizing common functionality. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

investigate the declarations of the newly added functions and see that they do not have a ROBOCOP signature. Hence, the major re-factoring performed in version 8 does not change the component interface but is rather an attempt to *factor out* common implementation code (C) in order to make the component code more readable.

6.5. Study conclusions

Two final issues emerged at the end of the ROBOCOP component framework assessment. First, our concrete findings, e.g. "component interfaces stayed unchanged for the following *n* versions", were confirmed by the developers as known, correct facts. Second and more interestingly, some findings led us to hypotheses, e.g. "the patterns seen here denote code re-factoring", that were unknown to the developers, but, at further detailed code inspection, were found correct. Thus, our visualizations let the users see known information and also discover new facts about a given component structure and implementation.

7. User study 3: Evolution of projects

As CVSscan helps in spotting correlations at line level, it clearly cannot display all lines in all files of a million lines of code, industry-size project. Typical limits for CVSscan are visualizing the evolution of one up to five files together. To show correlations across a larger set of files, we must use higher granularity entities than lines. CVSgrab uses the highest granularity level, i.e. one file = one entity, to support cross-file correlations across the evolution of entire projects.

In order to validate the visualization techniques proposed by CVSgrab, we organized a number of informal user studies. These studies record and analyze the experiences of software developers when they investigate new code bases with no other support than CVSgrab. Additionally, we compare their findings with the real state of affairs. The methodology and the promising outcome of such a study are described next.

Case study: Analysis of the VTK library

Three developers participated first in a 15 min training in which the functionality of CVSgrab was explained on a small example project containing the evolution of 28 files across up to 22 versions over 4 years, with several generic use cases that could be easily reproduced on other input data. Next, the users had to analyze an industry-size project and reason about their findings. In parallel, a fourth developer, with over seven years of development and use experience with the system under investigation, checked the findings of the three developers against his knowledge. To better simulate a real-life project assessment, the knowledgeable developer did not give any feedback during the study. At the end of the study, the experienced developer commented on the findings. The analyzed system was VTK, the industry-leader data visualization library of hundreds of C++ classes in over 2700 files, spanning over 100 versions, developed by more than 40 programmers over 10 years.

The users started CVSgrab in default mode, displaying the evolution of the entire VTK on one screen, (Fig. 20). Each file is represented as a horizontal blue pixel strip, divided into a number of segments by small bright yellow

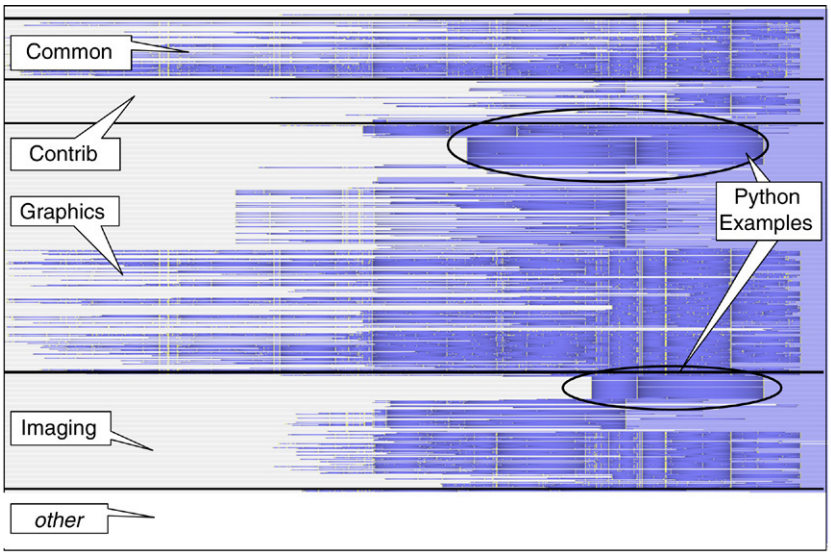

Fig. 20. VTK system evolution; files are displayed in the order in which they are available in the repository (i.e. implicitly grouped on folders). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

dots. The horizontal time axis has a uniform time sampling with one-second resolution. A version is thus implicitly represented as the blue horizontal segment delimited by two bright yellow commit events. Color encodes the version status: white = before creation of first version; dark blue = current version; light blue = after creation of last version. In Fig. 20, files are arranged on the vertical axis in the order they are stored in the CVS repository, thus is implicitly grouped on folders.

This layout is interesting for several reasons. First, it is able to compactly display hundreds up to thousands of files across hundreds of versions on a single screen with no or little scrolling. Second, it quickly lets users find the main directories, which appear as large, contiguous groups of files on the vertical axis. In our case, the users quickly identified that VTK consists of the following main directories: *Common*, *Contrib*, *Graphics* and *Imaging*. Since the latter two take the most Y-axis space (Fig. 20), they were correctly identified as containing the core of the VTK project. Third, it lets one quickly spot files with correlated evolutions, i.e. files that were created and changed at roughly the same time instants. These files appear as horizontal pixel strips having the bright commit events at roughly the same X-axis positions. If these files also share the same directory, the visual correlation is even stronger, as the pixel strips appear close to each other along the Y-axis. Two examples of this are visible in Fig. 20. These are the VTK Python examples in the graphics and imaging directories.

Cross-file evolution correlation, shown by the bright dotted vertical stripes of similar commit times, is clearly visible across the whole project, not just single directories. In our study, these stripes invariably caught the users' attention and one tried out to find more about the nature of these project-wide events. However, these stripes are not continuous, as files are sorted in the Y-axis according to directory location, not to commit similarity. Different file orders on the Y-axis can help this problem by putting files close to each other that have similar evolutions. The first alternative is to arrange the files according to creation time, starting from the assumption that files created at the same moment may have a similar evolution. The resulting visualization (Fig. 21) proved to be better for investigating the vertical commit event stripes.

By brushing with the mouse to get detailed information about these events from the commit log, the users discovered that most system-wide changes addressed copyright text modifications in all the files' heading sections. However, they detected also an important architectural change: the introduction of a new design pattern for a widely used type of static method. This visualization has a second advantage: It lets one quickly discover the moments when many of the files were suddenly introduced in the project, i.e. so-called moments of punctuated evolution. Four such moments are clearly visible in Fig. 21. A closer look at the file names led us conclude that only one such moment (highlight 3, Fig. 21) was related to the framework functionality, the other three (highlights 1, 2, 4, Fig. 21) being just massive additions of use examples. The second alternative for ordering files along the Y-axis uses the *activity*

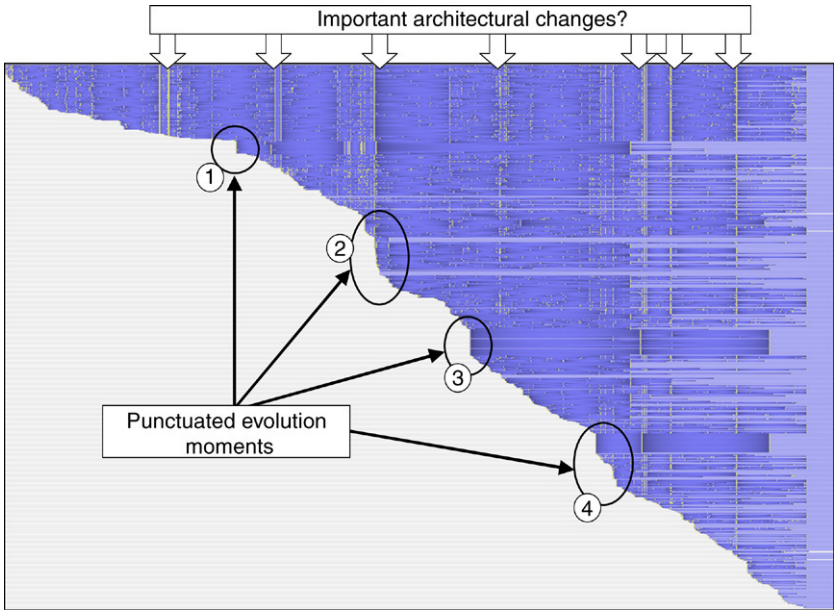

Fig. 21. VTK system evolution; files displayed in order of creation time.

Table 2
VTK developer network analysis

(1) = *schroede*	Qualified as the project initiator. His contribution seems however to stop about the middle of the project.
(2) = *martink*	The second major developer.
(3) = *lawcc*	More active towards the middle of the project.
(4) = *hoffman*	One major contribution in the middle of the project. From detailed information it appears he performed the major architectural change related to the introduction of a new type of static method, identified in Fig. 21.
(5) = *lorensen*	Started to contribute about the same time as *martink*. However, he becomes more active towards the middle of the project.
(6) = *will*	Started to contribute after the middle of the project. Apparently he replaces *schroede*.

measure, i.e. the total number of commits a file has, and is shown in Fig. 22. Here, files with highest activity are at the top of the visualization. This alternative proved to be also efficient in assessing the nature of the commit event vertical stripes associated with system-wide changes, especially in the later phases of the project. Additionally, this visualization brought new insights in the project evolution. The users identified a number of outliers, i.e. files whose commit activity did not match the average activity of the files created in the same period. We identified late outliers, i.e. files that have come later in the project but had a very high commit activity, and early outliers, i.e. files that have come early in the project but had a low commit activity. The users' assumption was that the early outliers corresponded mainly to stable interfaces and the late outliers to unstable implementations. Using the details-on-demand feature the users started to identify these outliers. As early outliers they identified mainly header files, e.g. `vtkRender.h`, `vtkCellType.h` `vtkMarchingCubesCases.h`, but also implementation files `vtkStack.cxx`, `vtkIndent.cxx`. As late outliers the users identified mainly implementation source code files, e.g. `vtkRenders.cxx`, `vtkPolyData.cxx`, `vtkImageData.cxx`, `vtkRenderWindow.cxx` but also paired header-implementation files, e.g. `vtkDataObject.cxx/.h`, which they correctly associated with architectural patches.

Next the users tried to get insight into the network of VTK developers, or authors. For this, they changed the color encoding to an author id-based one, i.e. all versions committed by the same author get the same unique color (similarly to Fig. 8). They also used creation time ordering along the vertical axis. From the result depicted in Fig. 23, they concluded the main work was done by just 6 of the more than 40 developers. They also identified their names. The concrete findings are presented in Table 2.

The three users further identified two other authors who appear to have a significant contribution to: (7) = *heiland* and (8) = *prabhu*. However, at a closer look using the details-on-demand mechanism, they concluded that these

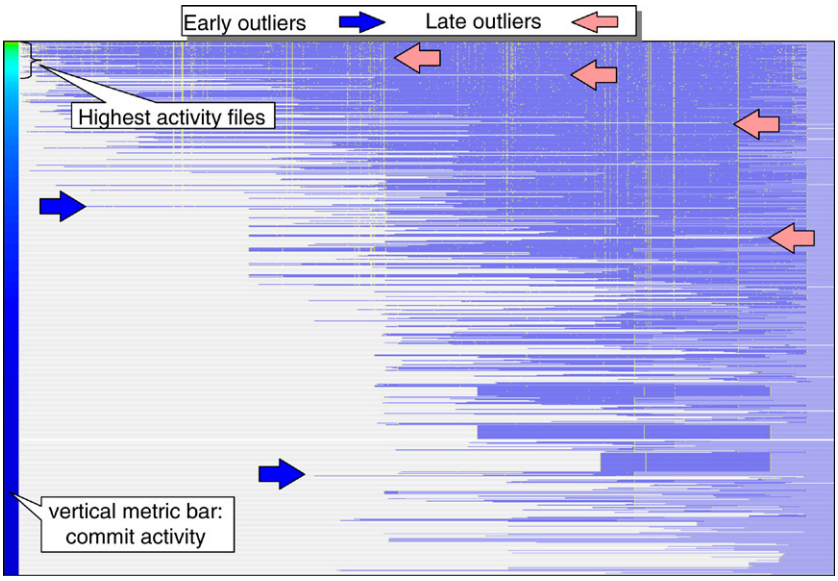

Fig. 22. VTK system evolution. Files are shown in the increasing order of activity.

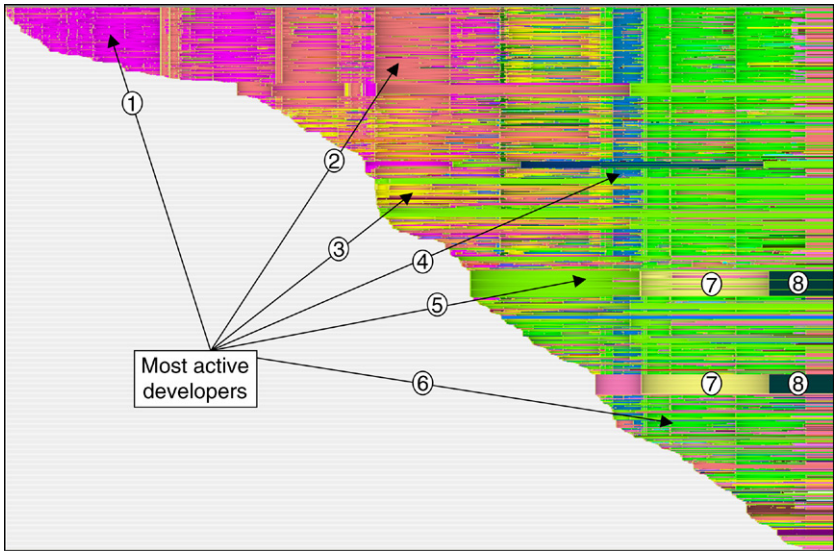

Fig. 23. VTK system evolution. Files are shown in order of creation time, color encodes author IDs. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

authors had done only small modifications, i.e. added a variable, in the library usage example files. They appear to be important because they modify many files.

7.1. Study conclusions

We stopped the study after 2 h. At the end, we summarized the three users' observations and checked them against the knowledge of the expert developer. The latter validated the largest part of the observations as fully correct. Moreover, he was particularly surprised by the ease with each the users identified the main people behind VTK, and the problematic/active areas of development, without *any prior knowledge* about this code. One aspect he found interesting was the higher-than-expected ratio between the 'project core' and the rest of the project activity. He identified also one misinterpretation concerning the authors. He declared that schroede and will refer actually to the same contributor (Will Schroeder, one of the parents of the VTK project) who changed his user name about mid-project. Still, this correctly matches with the users' observation that *will* replaces *schroede*.

8. Discussion

We have presented a set of techniques and tools for visually assessing the evolution of source code in large software projects. Our work builds on the main assumption that developers are most comfortable with visualizations that present the code in the same spatial context in which they construct it [10]. Therefore, our approach has the source code in focus, whenever this is possible (CVSscan). At lower, coarser, levels of detail, it is however not possible to visualize every line of code, so we keep the same visual metaphor but change the entity under scrutiny from code line to an entire file (CVSgrab). Our approach and code model (Section 3) is generic and can be applied to any type of source code.

CVSscan's line-based code model currently uses the UNIX-like `diff` used by the CVS repository itself to compare code. Although this makes CVSscan applicable to any type of source code, a weak point is the accuracy of the `diff` operator. The visualization accuracy depends on the heuristics behind this operator, which can lead to data misinterpretations, e.g. when too many changes occur between consecutive versions. However, our techniques and data model can support any user-provided `diff` operator. For example, one can use a syntactic, instead of line-based, code model, where the central entity is a component's interface, defined e.g. as a list of methods. Once a `diff` mechanism is provided for such a model, e.g. by interface comparison via method signatures, the visualization techniques presented here can be straightforwardly used. Instead of a code line, users would see a method, class, or component. The larger granularity of the entities would also make the methods applicable to larger projects. Yet another step in this direction would be to use a `diff` operator with support for semantic, instead of purely lexical or syntactic, comparisons. Note, however, that the existence of such more advanced `diff` operator possibilities does not invalidate the usefulness of simpler ones, such as the classical line-based one presented here. Simpler `diff` operators are fast, easy to implement, work on any source code or even plain text, and perform surprisingly well in many situations.

Another point of improvement in our work is the methodology of the user studies. All studies we have organized so far involved mainly users who were not familiar with the systems under investigation. In the first studies we organized (Section 5), we tried to involve users outside of the CVSscan and CVSgrab developers, not to bias the judgment by the expert tool-user point of view. However, we could only assess the validity of their findings using available means, as we were also not familiar with the systems under investigation. In the second type of studies we organized (Section 6), we used the knowledge of expert users to verify the validity of the findings, but we performed the studies inside our development community, which has extensive expertise with the tools. In the last studies we performed (Section 7) we involved both users outside our community and an expert user to verify the validity of their findings at the end. Still, the users performed the investigation as a team, so they benefited also from the intense discussions they had. For the future, we would like to organize user studies with more users outside our community. Additionally, we would like the users to carry on the studies without interaction, and eventually to have their findings always validated by an expert.

9. Conclusion

We have presented a new approach for the visualization of large-scale software evolution using code-oriented displays. We presented a generic code data model that can describe change at various levels of granularity. Next, we introduced several visual mappings and interaction techniques for efficiently and effectively visualizing this code model. All work presented here was implemented by two tools developed by us, CVSscan and CVSgrab, in order to validate the proposed techniques. These tools, as well as several example source code datasets, are available for download at http://www.win.tue.nl/~lvoinea/VCN.html.

The main audience we target with our work is the software developer and maintenance community, ranging from programmers to system architects and technical project managers. The goal is to provide them with effective support for program and process understanding by exploiting the evolution information contained in SCM repositories such as CVS or Subversion.

Our novel approach uses multiple correlated views on the evolution of a software project. We use dense pixel displays to show the overall evolution of code structure and attributes of tens of thousands of elements on a single screen, and we integrate them in an orchestrated environment to offer details-on-demand.

We have also presented in this paper the typical outcome of a number of user studies we did to validate our approach on data from real-life, industry-size CVS repositories. Although informal, the studies show that our proposed

code-based evolution visualization of software supports a quick assessment of the important activities and artifacts produced during development, even for users that had not taken part in any way in developing the examined projects. The subjects of our user studies valued most the compact overview coupled with easy access to detailed information such as source code or developer comments. These enabled them to easily spot issues at a high level and then get detailed information for further refinement of their assessments. An extremely important acceptance point was the simplicity of use of our tools: To produce most of the visualizations shown here, or similar ones, one needs only to start the tool, type in the location of a CVS repository, and wait for the screen to be populated with visual information about the downloaded data. Most subsequent manipulations, such as sorting entities, changing color attributes, getting details on demand and so on, are reachable via just a few mouse clicks.

In the future, we plan to extend and refine our set of methods and techniques for visual code evolution investigation in two main directions. First, we work to refine the code data model to incorporate several higher-level abstractions (classes, methods, namespaces) and their corresponding `diff` operators. Second, we are actively researching novel ways to display the existing information in more compact, more suggestive ways. We plan to actively conduct user tests to assess the concrete value and effectiveness of such visualizations, which is the ultimate proof of our proposed techniques.

References

[1] Articus Systems, Rubus OS — reference manual, 1996.

[2] T. Ball, J.-M. Kim, A.A. Porter, H.P. Siy, If your version control system could talk..., in: ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering, May 1997. Available online at: http://research.microsoft.com/~tball/papers/icse97-decay.pdf.

[3] T. Ball, S. Eick, Software visualization in the large, IEEE Computer 29 (4) (1996) 33–43.

[4] K. Beck, C. Andres, Extreme Programming Explained: Embrace Change, 2nd ed., Addison-Wesley, 2000.

[5] J.M. Bieman, A.A. Andrews, H.J. Yang, Understanding change-proneness in OO software through visualization, in: Proc. 11th International Workshop on Program Comprehension, IEEE CS Press, Washington, DC, USA, 2003, pp. 44–53.

[6] C. Burrows, I. Wesley, Ovum Evaluates: Configuration Management, Ovum Inc., Burlington, MA, USA, 1999.

[7] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, K. Wampler, A system for graph-based visualization of the evolution of software, in: Proc. ACM SoftVis '03, ACM Press, NY, USA, 2003, pp. 77–86.

[8] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, 16th ed., MIT Press, 1996.

[9] CVS online: http://www.nongnu.org/cvs/.

[10] S.G. Eick, J.L. Steffen, E.E. Sumner, Seesoft — a tool for visualizing line oriented software statistics, IEEE Transactions on Software Engineering 18 (11) (1992) 957–968.

[11] M. Eiglsperger, M. Kaufmann, M.A. Siebenhaller, Topology-shape-metrics approach for the automatic layout of UML class diagrams, in: Proc. ACM SoftViz '03, ACM Press, NY, USA, 2003, pp. 189–198.

[12] L. Erlikh, Leveraging legacy system dollars for e-business, IEEE IT Pro (2000) 17–23.

[13] M. Fischer, M. Pinzger, H. Gall, Populating a Release History Database from version control and bug tracking systems, in: Proc. International Conference on Software Maintenance, ICSM 2003, IEEE CS Press, Washington, DC, USA, 2003, pp. 23–32.

[14] FreeBSD online: http://www.freebsd.org/.

[15] J. Froehlich, P. Dourish, Unifying artifacts and activities in a visual tool for distributed software development teams, in: Proc. ICSE '04, IEEE CS Press, Washington, DC, USA, 2004, pp. 387–396.

[16] H. Gall, M. Jazayeri, J. Krajewski, CVS release history data for detecting logical couplings, in: Proc. IWPSE 2003, IEEE CS Press, Washington, DC, USA, 2003, pp. 13–23.

[17] H. Gall, M. Jazayeri, C. Riva, Visualizing software release histories: The use of color and third dimension, in: Proc. of the IEEE International Conference on Software Maintenance, ICSM'99, IEEE CS Press, p. 99.

[18] D. German, A. Mockus, Automating the measurement of open source projects, in: ICSE '03 Workshop on Open Source Software Engineering, Portland, Oregon, USA, May 2003. Available online at: http://www.research.avayalabs.com/user/audris/papers/oose03.pdf.

[19] T. Girba, Modeling History to Understand Software Evolution, Ph.D. Thesis. Available online at http://www.iam.unibe.ch/~girba/download/TGirba-PhD-Book.pdf.

[20] W.G. Griswold, J.J. Yuan, Y. Kato, Exploiting the map metaphor in a tool for software evolution, in: Proc. ICSE '01, IEEE CS Press, Washington, DC, USA, 2001, pp. 265–274.

[21] C. Gutwenger, M. Junger, K. Klein, J. Kupke, S. Leipert, P. Mutzel, A new approach for visualizing UML class diagrams, in: Proc. ACM SoftViz '03, ACM Press, NY, USA, 2003, pp. 179–188.

[22] ITEA, ROBOCOP: Robust Open Component Based Software Architecture for Configurable Devices Project—Framework concepts, Public Document V1.0, May 2002. Available online at: http://www.hitech-projects.com/euprojects/robocop/.

[23] J.A. Jones, M.J. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proc. ICSE '02, ACM Press, NY, USA, 2002, pp. 467–477.

[24] M. Lanza, The evolution matix: Recovering software evolution using software visualization techniques, in: Proc. International Workshop on Principles of Software Evolution, 10–11 September, 2001, Vienna, Austria, ACM Press, NY, USA, 2001, pp. 37–42.

[25] G. Lommerse, F. Nossin, S.L. Voinea, A. Telea, The visual code navigator: An interactive toolset for source code investigation, in: Proc. IEEE InfoVis'05, IEEE CS Press, Washington, DC, USA, 2005, pp. 24–31.
[26] L. Lopez-Fernandez, G. Robles, J.M. Gonzalez-Barahona, Applying social network analysis to the information in CVS repositories, in: International Workshop on Mining Software Repositories (MSR), Edinburgh, Scotland, UK, May 2004. Available online at: http://opensource.mit.edu/papers/llopez-sna-short.pdf.
[27] A. Marcus, L. Feng, J.I. Maletic, 3D representations for software visualization, in: Proc. ACM SoftVis '03, ACM Press, NY, USA, 2003, pp. 27–36.
[28] A. Mockus, R.T. Fielding, J.D. Herbsleb, Two case studies of open source software development: Apache and Mozilla, ACM Transactions on Software Engineering and Methodology 11 (3) (2002) 309–346.
[29] A. Mller, M. Kerholm, J. Fredriksson, M. Nolin, Evaluation of component technologies with respect to industrial requirements, in: Proc. of the 30th EUROMICRO Conference, EUROMICRO'04, 31 August–3 September 2004, Rennes, France, IEEE CS Press, Washington, DC, USA, 2004, pp. 56–63.
[30] M. Ohira, N. Ohsugi, T. Ohoka, K. Matsumoto, Accelerating cross project knowledge collaboration using collaborative filtering and social networks, in: Proc. International Workshop on Mining Software Repositories, MSR, Saint Louis, Missouri, USA, May 2005, pp. 111–115. Available online at: http://plg.uwaterloo.ca/~aeehassa/home/pubs/MSR2005ProceedingsFINAL_ACM.pdf.
[31] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, The Koala component model for consumer electronics, IEEE Transactions on Computers 33 (3) (2000) 78–85.
[32] S.P. Reiss, Bee/Hive: A software visualization back end, in: Workshop on Software Visualization, ICSE '01, 2001. Available online at: http://www.cs.brown.edu/~spr/research/bloom/beehive.pdf.
[33] R.C. Seacord, D. Plakosh, G.A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Process, and Business Practices, in: SEI Series in Software Engineering, Addison-Wesley, 2003.
[34] B. Shneiderman, The eyes have it: A task by data type taxonomy for information visualization, in: Proc. IEEE Symp. on Visual Languages, VL '96, IEEE CS Press, Washington, DC, USA, 1996, pp. 336–343.
[35] J. Sliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? On Fridays, in: Proc. International Workshop on Mining Software Repositories, MSR, Saint Louis, Missouri, USA, May 2005, pp. 24–28. Available online at: http://plg.uwaterloo.ca/~aeehassa/home/pubs/MSR2005ProceedingsFINAL_ACM.pdf.
[36] M.A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, M. Musen, SHriMP views: An interactive environment for information visualization and navigation, in: Proc. CHI '02, ACM Press, NY, 2002, pp. 520–521.
[37] Subversion online: http://subversion.tigris.org/.
[38] A. Telea, A. Maccari, C. Riva, An open toolkit for prototyping reverse engineering visualization, in: Proc. IEEE VisSym '02, The Eurographics Association, Aire-la-Ville, Switzerland, 2002, pp. 241–251.
[39] S.R. Tilley, K. Wong, M.A.D. Storey, H.A. Muller, Rigi: A visual tool for understanding legacy systems, International Journal of Software Engineering and Knowledge Engineering (1994).
[40] F. van Rysselberghe, S. Demeyer, Studying software evolution information by visualizing the change history, in: Proc. 20th IEEE International Conference on Software Maintenance, ICSM'04, IEEE CS Press, 2004, pp. 328–337.
[41] J. van Wijk, H. van de Wetering, Cushion treemaps: Visualization of hierarchical information, in: Proc. IEEE InfoVis'99, IEEE CS Press, 1999, pp. 73–78.
[42] L. Voinea, A. Telea, J.J. van Wijk, CVSscan: Visualization of code evolution, in: Proc. of the ACM Symposium on Software Visualization, SoftVis'05, ACM Press, NY, USA, 2005, pp. 47–56.
[43] L. Voinea, A. Telea, M. Chaudron, Version centric visualization of code evolution, in: Proc. of the IEEE Eurographics Symposium on Visualization, EUROVIS'05, IEEE CS Press, 2005, pp. 223–230.
[44] VTK online: http://www.kitware.com/.
[45] A.T.T. Ying, G.C. Murphy, R. Ng, M.C. Chu-Carroll, Predicting source code changes by mining revision history, IEEE Transactions on Software Engineering 30 (9) (2004) 574–586.
[46] M. Winter, T. Genssler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arvalo, P. Mller, C. Stich, B. Schnhage, Components for embedded software—the Pecos approach, in: Second International Workshop on Composition Languages in conjunction with 16th European Conference on Object-Oriented Programming, ECOOP, Malaga, Spain, June 11, 2002. Available online at: http://www.iam.unibe.ch/~scg/Archive/pecos/public_documents/Wint02a.pdf.
[47] J. Wu, C.W. Spitzer, A.E. Hassan, R.C. Holt, Evolution spectrographs: Visualizing punctuated change in software evolution, in: Proc. 7th International Workshop on Principles of Software Evolution, IWPSE'04, IEEE CS Press, Washington, DC, USA, 2004, pp. 57–66.
[48] T. Zimmermann, S. Diehl, A. Zeller, How history justifies system architecture (or not), in: Proc. IWPSE 2003, IEEE CS Press, Washington, DC, USA, 2003, pp. 73–83.
[49] T. Zimmermann, P. Weigerber, S. Diehl, A. Zeller, Mining version histories to guide software changes, in: Proc. 26th International Conference on Software Engineering, ICSE, May 2004, Edinburgh, Scotland, IEEE Press Piscataway, NJ, USA, 2004, pp. 429–445.
[50] T. Zimmermann, P. Weigerber, Preprocessing CVS data for fine-grained analysis, in: International Workshop on Mining Software Repositories, MSR, Edinburgh, Scotland, UK, May 2004. Available online at: http://www.st.cs.uni-sb.de/papers/msr2004/msr2004.pdf.